Chapter 6

Separation Logic

Separation logic was introduced around year 2000 by Reynolds and O'Hearn [6]. There are indeed several variants of this logic [1, 8]. It is implemented in several tools like Smallfoot [2] and Verifast [7], that use specific, partially automated provers as back-ends, or Ynot [5] or CFML [3] that uses Coq to perform the proofs.

6.1 Basics of Separation Logic

6.1.1 Syntax of Programs

We do not consider any encoding of pointers and memory heap like the Component-as-Array model of Section 5.2.3, so we directly consider the language introduced in Section 5.2.1. But, to make the formalization of memory access uniform, we remove mutable variables from that language: every mutable data must now be of the form of a pointer to a record, and only record fields are mutable. This is indeed how the references in OCaml or in Why3 are introduced: these are not built-in in the language but defined using a record with one mutable field. We detail this in Section 6.2.

For declaring such records, we keep the same syntax as in previous chapter.

$$\mathsf{record}\ id = \{f_1 : \tau_1; \cdots f_n : \tau_n\}$$

We also generalize this language in order to deal with memory allocation and deallocation. The important point is how to extend the language of formulas in specifications, which is detailed in Section 6.1.3.

The syntax of expressions is now as follows.

e	::=	...	former expression constructs (without mutable variables)
		$\mid e \to f$	field access
		$\mid e \to f := e$	field update
		$\mid$ `new` S	allocation
		$\mid$ `dispose` e	deallocation

The expression `new` S allocates a new record of type S, and returns a pointer to it. The expression `dispose` e deallocates the pointer denoted by expression e, and returns $()$.

6.1.2 Operational Semantics

To support allocation and deallocation, we modify the operational semantics given in Section 5.2.2. Instead of considering the memory heap as a *total* map from pairs (loc,field name) to values, we consider

that a heap is any *partial* map from (loc,field name) to values. We need to introduce a few notations: if h, h_1, h_2 are such partial maps:

- $\mathsf{dom}(h)$ denotes the *domain* of h, i.e. the set of pairs (loc,field name) where it is defined ;
- we write $h = h_1 \oplus h_2$ to mean that h is the disjoint union of h_1 and h_2, i.e.
  - $\mathsf{dom}(h_1) \cap \mathsf{dom}(h_2) = \emptyset$
  - $\mathsf{dom}(h) = \mathsf{dom}(h_1) \cup \mathsf{dom}(h_2)$
  - $h(l, f) = h_1(l, f)$ if $(l, f) \in \mathsf{dom}(h_1)$
  - $h(l, f) = h_2(l, f)$ if $(l, f) \in \mathsf{dom}(h_2)$

Note that in the following, the domain of partial heaps are always finite.

The operational semantics is defined by the relation

$$h, \Pi, e \rightsquigarrow h', \Pi', e'$$

where h is a partial heap. The rules are as follows.

Field access

$$\frac{l, f \in \mathsf{dom}(h) \qquad h(l, f) = v}{h, \Pi, (l \to f) \rightsquigarrow h, \Pi, v}$$

Implicitly, this rule means that if $(l, f) \notin \mathsf{dom}(h)$ then the execution cannot be done, it "blocks". Typically in practice this corresponds to an invalid memory access, which usually stops the program execution with a message like "segmentation fault" or "memory fault"

Field update

$$\frac{(l, f) \in \mathsf{dom}(h) \qquad h' = h\{(l, f) \leftarrow v\}}{h, \Pi, (l \to f := v) \rightsquigarrow h', \Pi, ()}$$

Again, if $(l, f) \notin \mathsf{dom}(h)$ then the execution blocks.

Allocation

$$\frac{l \notin \mathsf{dom}(h) \qquad h' = h \oplus \{(l, f) \leftarrow \mathit{default}(\tau) \mid f : \tau \in S\}}{h, \Pi, (\texttt{new}\ S) \rightsquigarrow h', \Pi, l}$$

The premise $l \notin \mathsf{dom}(h)$ means that l is "fresh", i.e. it is not yet allocated in h. The other premise expresses the initialization of the allocated fields depending on their types, i.e. $\mathit{default}(int) = 0$, $\mathit{default}(bool) = \mathit{false}$, $\mathit{default}(real) = 0.0$, $\mathit{default}(S) = null$.

Notice that from this definition, the allocation never blocks, our formalization does not consider memory overflow issues.

Deallocation

$$\frac{\text{for all } f \text{ of } S, (l, f) \in \mathsf{dom}(h) \qquad h' = h \backslash l}{h, \Pi, (\texttt{dispose}\ l) \rightsquigarrow h', \Pi, ()}$$

where the notation $h' = h \backslash l$ means $h'(l', f') = h(l', f')$ if $l' \neq l$, undefined otherwise. Notice that deallocation blocks if one attempts to deallocate a memory location that is not allocated.

Example 6.1.1 *The small program below illustrates our operational semantics.*

```
record List = { data : int, next: List }
```

$[], []$
`let x = new List in` $\rightsquigarrow$
$[(l_0, data) = 0, (l_0, next) = null], [x = l_0]$
`x→next := new List;` $\rightsquigarrow$
$[(l_0, data) = 0, (l_0, next) = l_1, (l_1, data) = 0, (l_1, next) = null], [x = l_0]$
`dispose(x→next);` $\rightsquigarrow$
$[(l_0, data) = 0, (l_0, next) = l_1], [x = l_0]$
`x→next→data := 1`
execution blocks

6.1.3 Separation Logic Formulas

A important idea in Separation Logic is that the language of terms remains *unchanged*, in particular the expression $e \rightarrow f$ *is not a term*. The language of formulas is extended with

- special atoms specifying *available memory ressources*
- a special connective called *separating* conjunction

The new grammar for formulas is as follows.

$$\begin{array}{rcll} P, Q & ::= & \ldots & \text{former constructs} \\ & & \mid \mathsf{emp} & \text{empty heap} \\ & & \mid t_1 \overset{f}{\mapsto} t_2 & \text{memory chunk} \\ & & \mid P * Q & \text{separating conjunction} \end{array}$$

where

- t_1 is a term of type S for some record type S
- f is a field of type τ in S
- t_2 is a term of type τ

These three new constructs allow to describe *finite portions of the memory heap*

The semantics of formulas is defined as usual by an interpretation $[\![P]\!]_{h,\Pi}$ that now depends on a partial heap h.

- Special formula emp:

$$[\![\mathsf{emp}]\!]_{h,\Pi} \text{ valid iff } \mathsf{dom}(h) = \emptyset$$

- Memory chunk: $[\![t_1 \overset{f}{\mapsto} t_2]\!]_{h,\Pi}$ iff
  - $[\![t_1]\!]_\Pi = l$ for some location l
  - $\mathsf{dom}(h) = \{(l, f)\}$
  - $h(l, f) = [\![t_2]\!]_\Pi$
- Separating conjunction: $[\![P * Q]\!]_{h,\Pi}$ is valid iff
  there exists h_1, h_2 such that

- $h = h_1 \oplus h_2$
- $[\![P]\!]_{h_1,\Pi}$ is valid
- $[\![Q]\!]_{h_2,\Pi}$ is valid

The semantics of the separating conjunction explicitly requires that the two conjuncts are valid on disjoint heaps. This means that predicates like our former predicates `disjoint` and `no_repet` are in some sense *internalized* in the logic.

Example 6.1.2 *Assuming two non-null locations l_0 and l_1, the stack $\Pi = [x = l_0]$ and the following partial heaps*

$$\begin{aligned} h_1 &= [(l_0, next) = l_1] \\ h_2 &= [(l_0, next) = l_1, (l_0, data) = 42] \\ h_3 &= [(l_0, next) = l_1, (l_1, next) = null] \end{aligned}$$

the table below gives the validity of a few formulas (Y for valid, N otherwise).

valid in ?	h_1	h_2	h_3
$x \overset{next}{\mapsto} l_1$	Y	N	N
$x \overset{next}{\mapsto} l_1 * x \overset{data}{\mapsto} 42$	N	Y	N
$x \overset{next}{\mapsto} l_1 * l_1 \overset{next}{\mapsto} null$	N	N	Y
emp	N	N	N

Properties of Separating Conjunction Separating conjunction satisfies the following identities

- $(P * Q) * R \leftrightarrow P * (Q * R)$
- $P * Q \leftrightarrow Q * P$
- $\mathsf{emp} * P \leftrightarrow P$
- if P, Q are memory-free formulas, $P * Q \leftrightarrow P \wedge Q$

In general P not equivalent to $P * P$. In fact this identity is true only if P is emp or a memory-free formula. This is a *linearity* aspect of the separating conjunction.

Symbolic Heaps The semantics given above has some issues regarding memory-free formulas. For example, using the heaps of Example 6.1.2 above, even if the formula $x \overset{next}{\mapsto} l_1$ is valid only in heap h_1, the conjunction $x \overset{next}{\mapsto} l_1 * true$ is valid also in heap h_1 and h_2. A variant in the semantics would be to consider that a memory-free formula is valid only in an empty heap.

Going further, a classical fragment of separation logic is the so-called *Symbolic Heaps*. Only formulas of the form

$$\exists v_1, \ldots, v_n, P_1 * P_2 * \cdots * P_k * \phi$$

are considered, where ϕ is a memory-free formula and the P_i are memory chunks.

In the rest of this chapter, we consider this fragment only, with the semantics that memory-free formulas are valid only in an empty heap.

Example 6.1.3 *The following function resets the field `f` of some structure to zero.*

```
record S { f : int }

function reset_f(x:S):unit
  requires  ∃v.x ↦f v
  ensures   x ↦f 0
  body x→f := 0
```

Since one cannot directly use `x→f` as a term, an extra quantification is needed to specify that `x→f` is allocated.

The following function increments the field `f` by one.

```
function incr_f(x:S) (ghost v:int):unit
  requires  x ↦f v
  ensures   x ↦f v + 1
  body x→f := x→f + 1
```

A ghost variable is needed to talk about the old value of `x→f`.

Some syntactic sugar are usually used: existentially quantified variables starting with an underscore, and quantification made implicit.

Example 6.1.4 *The same functions as in previous example, with syntactic sugar*

```
function reset_f(x:S):unit
  requires  x ↦f _
  ensures   x ↦f 0
  body x→f := 0

function incr_f(x:S) :unit
  requires  x ↦f _v
  ensures   x ↦f _v + 1
  body x→f := x→f + 1
```

Notice how the ghost variable is made implicit.

6.1.4 Separation Logic Reasoning Rules

We can define rules for reasoning on programs in the similar style as Hoare rules, on triples $\{P\}e\{Q\}$ where P and Q are symbolic heaps.

The following rules can easily be proved correct.

Field access

$$\frac{}{\{l \overset{f}{\mapsto} v\}l \to f\{l \overset{f}{\mapsto} v * \mathsf{result} = v\}}$$

Field update

$$\frac{}{\{l \overset{f}{\mapsto} v\}l \to f := v'\{l \overset{f}{\mapsto} v' * \mathsf{result} = ()\}}$$

Allocation

$$\frac{}{\{\mathsf{emp}\}\texttt{new}\ S\{\_l \overset{f_1}{\mapsto} default(\tau_1) * \cdots * \_l \overset{f_n}{\mapsto} default(\tau_n) * \mathsf{result} = \_l\}}$$

where S is declared as $\{f_1 : \tau_1; \cdots f_n : \tau_n\}$

Deallocation

$$\frac{}{\{l \overset{f_1}{\mapsto} v_1 * \cdots * l \overset{f_n}{\mapsto} v_n\}\texttt{dispose}\ l\{\mathsf{emp}\}}$$

where l has type S.

Frame rule A very important rule that can be stated is the frame rule, which allows to reason *locally*:

$$\frac{\{P\}e\{Q\}}{\{P * R\}e\{Q * R\}}$$

in particular it states that whenever $\{P\}e\{Q\}$ is proven valid, its effects are confined in the partial heaps that are described by P and Q, and consequently it remains valid in any context R which is disjoint from P and Q.

Separation Logic and Symbolic Execution It has been noticed that reasoning with Separation Logic rules is a kind of *symbolic execution* [1]. Thanks to the frame rule, a proof of a program made of a sequence $e_1; \cdots; e_n$ can be presented under the form

$$\{P_0\}e_1\{P_1\}e_2 \cdots e_n\{P_n\}$$

where at each step i we have $P_{i-1} = Q * R$, $P_i = Q' * R$ and $\{Q\}e_i\{Q'\}$ is valid for a rule above.

Example 6.1.5 *Here is a simple example of Separation Logic proof in the form of symbolic execution.*

$\{emp\}$
let *x = new List* ***in***
$\{(\_l \overset{data}{\mapsto} 0) * (\_l \overset{next}{\mapsto} null) * (x = \_l)\}$
$\{(x \overset{data}{\mapsto} 0) * (x \overset{next}{\mapsto} null)\}$ *(consequence)*
x→data := 42 ;
$\{(x \overset{data}{\mapsto} 42) * (x \overset{next}{\mapsto} null)\}$ *(frame)*
x→next := new List ;
$\{(x \overset{data}{\mapsto} 42) * (x \overset{next}{\mapsto} \_l) * (\_l \overset{data}{\mapsto} 0) * (\_l \overset{next}{\mapsto} null)\}$

6.2 Case of References à la OCaml

The mutable variables that we had in previous chapters but we dropped in our new language, can be simulated using a pointer to a record with one field only. For convenience it is made polymorphic.

```
record Ref α = { contents : α }

function ref (x: α) : Ref α
  body let r = new Ref in r→contents := x; r
```

```
function (!) (r: Ref α) : α
  body r→contents

function (:=) (r: Ref α) (x:α) : unit
  body r→contents := x
```

These functions can be given natural specifications in separation logic. For readability we abbreviate $r \overset{contents}{\mapsto} t$ into $r \mapsto t$

```
function ref (x: α) : Ref α
  requires  emp
  ensures   result ↦ x

function (!) (r: Ref α) : α
  requires  r ↦ _v
  ensures   r ↦ _v ∗ result = _v

function (:=) (r: Ref α) (x:α) : unit
  requires  r ↦ _
  ensures   r ↦ x
```

6.3 Case of Linked Lists

Similarly as the case of Component-as-Array model, we can define linked data structures using inductive predicates. For linked list, we can define

inductive $ls(List, List)$ =
| ls_nil: $\forall x : List, ls(x, x)$
| ls_cons: $\forall xyz : List, (x \overset{next}{\mapsto} y) * ls(y, z) \rightarrow ls(x, z)$

The important novelty is the use of a separating conjunction in the second case, which makes explicit that the tail of the list is disjoint from its head, and thus it has no repetition.

During a Separation Logic proof, one should apply purely logic reasoning steps using general lemmas like

$$ls(x, y) \leftrightarrow x = y \vee \exists z, (x \overset{next}{\mapsto} z) * ls(z, y)$$

From such lemmas one can derives the so-called *symbolic execution rules*, e.g.

$$ls(x, y) * x \neq y \rightsquigarrow (x \overset{next}{\mapsto} \_z) * ls(\_z, y) * x \neq y$$

In practice, systems implementing separation logic use a specific reasoning engine that applies such rules (they do not use off-the-shelf SMT provers).

Example: in-place list reversal

We consider again the example of in-place linked list reversal. We simplicity, we do not specify that the result list is the reverse of the input, but only that the output is a null-terminated list if the input is such a list. This avoids the need for ghost variables. The specified program we want to prove is thus the following (we use our "references" defined in previous section).

```
function reverse (l:List) : List =
  requires ls(l,null)
  ensures ls(result,null)
  body
   let p = ref l in
   let r = ref null in
   while !p ≠ null do
     invariant p→_lp * ls(_lp,null) * r→_lr * ls(_lr,null)
     let n = !p→next in
     !p→next := r;
     r := !p;
     p := n;
  done;
  !r
```

Notice that the loop invariant tells that lists $!p$ and $!r$ are null-terminated and disjoint.

The first part of the proof is to show by symbolic execution that the loop invariant is initially true. This is as follows.

$\{ls(l, null)\}$
`let p = ref l in`
$\{p \mapsto l * ls(l, null)\}$
`let r = ref null in`
$\{p \mapsto l * ls(l, null) * r \mapsto null\}$
$\{p \mapsto l * ls(l, null) * r \mapsto null * ls(null, null)\}$ (symbolic execution rule)
`while p ≠ null do`
`  invariant p→_lp * ls(_lp,null) * r→_lr * ls(_lr,null)`

The second part is to show the loop invariant is preserved by a loop iteration.

`while !p ≠ null do`
`  invariant p→_lp * ls(_lp,null) * r→_lr * ls(_lr,null)`
$\{p \mapsto l_p * ls(l_p, null) * r \mapsto l_r * ls(l_r, null) * l_p \neq \texttt{null}\}$
$\{p \mapsto l_p * l_p \overset{next}{\mapsto} \_q * ls(\_q, null) * r \mapsto l_r * ls(l_r, null) * l_p \neq \texttt{null}\}$
`  let n = !p→next in`
$\{p \mapsto l_p * l_p \overset{next}{\mapsto} n * ls(n, null) * r \mapsto l_r * ls(l_r, null) * l_p \neq \texttt{null}\}$
`  !p→next := !r;`
$\{p \mapsto l_p * l_p \overset{next}{\mapsto} l_r * ls(n, null) * r \mapsto l_r * ls(l_r, null) * l_p \neq \texttt{null}\}$
`  r := !p;`
$\{p \mapsto l_p * l_p \overset{next}{\mapsto} l_r * ls(n, null) * r \mapsto l_p * ls(l_r, null) * l_p \neq \texttt{null}\}$
`  p := n;`
$\{p \mapsto n * l_p \overset{next}{\mapsto} l_r * ls(n, null) * r \mapsto l_p * ls(l_r, null) * l_p \neq \texttt{null}\}$
$\{p \mapsto n * ls(n, null) * r \mapsto l_p * ls(l_p, null)\}$

Finally, we prove that when we exit the loop, the post-condition is established.

`while !p ≠ null do`
`  invariant p→_lp * ls(_lp,null) * r→_lr * ls(_lr,null)`
`  ...`
`done`
$\{p \mapsto l_p * ls(l_p, null) * r \mapsto l_r * ls(l_r, null) * l_p = \texttt{null}\}$

$\{p \mapsto null * r \mapsto l_r * ls(l_r, null)\}$
`!r`
$\{ls(result, null)\}$ (implicit dispose!)

6.4 Going Further on Separation Logic

The main motivation of Separation Logic is that it internalizes disjointness and frame properties. It makes reasoning on pointer data structures much easier than a model like the Component-as-Array, where disjointness and frame properties must be stated explicitly.

Separation Logic has several applications and extensions. An important application is to the specification of *data invariants*, that are properties that should remain true along the execution of a program (such as "this list should always contain non-negative integers", "this tree should always be balanced"). Separation Logic provides for free that a data invariant is preserved when memory is modify outside its frame. Separation Logic can also deals with *concurrent programs* [4], for example two threads executing in parallel on disjoint part of the memory can be proved easily.

A major drawback in Separation Logic techniques nowadays is their low level of automation. There no simple equivalent of weakest precondition calculus, and thus SMT solvers cannot be used directly, one has to design provers that can understand the separating conjunction. Indeed, some analogue to WP can be constructed, if one considers also a connective for *separating implication*, also known as *magic wand*[1].

6.5 Exercises

Exercise 6.5.1 *Prove a complete specification of linked list reversal via Separation Logic, using appropriate ghost variables and an extended predicate* `ls` *that includes the model list.*

Exercise 6.5.2 *Specify and prove the function of Exercise 5.3.3 which increments by one each element of a null-terminated linked list of integers, using Separation Logic.*

Exercise 6.5.3 *Specify and prove the function of Exercise 5.3.4 which appends two lists, using Separation Logic.*

Bibliography

[1] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In K. Yi, editor, *APLAS*, volume 3780 of *Lecture Notes in Computer Science*, pages 52–68. Springer, 2005.

[2] J. Berdine, C. Calcagno, and P. W. O'hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *In International Symposium on Formal Methods for Components and Objects*, pages 115–137. Springer, 2005.

[3] A. Charguéraud. Characteristic formulae for the verification of imperative programs. In M. M. T. Chakravarty, Z. Hu, and O. Danvy, editors, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming (ICFP)*, pages 418–430, Tokyo, Japan, September 2011. ACM.

[4] X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In R. D. Nicola, editor, *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 173–188. Springer, 2007.

[1] See `http://www.lsv.ens-cachan.fr/~lozes/sl-lectures.php`

[5] A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in Hoare type theory. In J. H. Reppy and J. L. Lawall, editors, *11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006*, pages 62–73, Portland, Oregon, USA, 2006. ACM Press. ISBN 1-59593-309-3.

[6] J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *17h Annual IEEE Symposium on Logic in Computer Science*. IEEE Comp. Soc. Press, 2002.

[7] J. Smans, B. Jacobs, F. Piessens, and W. Schulte. An automatic verifier for java-like programs based on dynamic frames. In *Fundamental Approaches to Software Engineering (FASE'08)*, Budapest, Hungary, Apr. 2008.

[8] H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In M. Hofmann and M. Felleisen, editors, *Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, pages 97–108, Nice, France, Jan. 2007.